Prolog Handout 1
Introduction to Prolog

Prolog stands for Programming in Logic. In prolog you specify a database of “truths” (facts and rules which specify generalizations and definitions). These things can be thought of as the axioms to a theorem prover. Next you ask some questions – does some statement follow from the facts previous defined. Prolog goes about attempting to prove that the query statement can be established to hold using the facts and rules. The interpreter does backward chaining in order to prove that a goal (statement to be established) follows from the axioms. The statement that you are asking about, typically has variables in it. During the proof, variables may become instantiated. If the proof succeeds, then the variable bindings are returned. Thus, as a side effect, the instantiation of variables may cause “computation” to be done.

Both the set of axioms (the program) and the query have a special form of formulae in predicate logic. A Horn clause is a clause (set of literals) with at most one positive literal. For now, you can think of a clause as a bunch of positive and negative literals. A positive literal, also called atomic formula, has the form \( p(t_1, \ldots, t_n) \), where \( p \) is a predicate symbol and \( t_i \) is a term. A negative literal is simply the negation of a positive literal.

In prolog, we consider three kinds of Horn clauses. A Horn clause with one or more negative literal but no positive literal is called a query. A Horn clause with exactly one positive literal can be a rule or a fact. The latter is a clause with no negative literals and a rule is a Horn clause with one positive literal and one or more negative literal.

Programming in Prolog – setting up a database and asking questions of it.

- declare some facts about objects and their relationships.
- define some rules about objects and their relationships.
- ask questions about objects and relationships.

**Facts** – simple predicate statements followed by a “.”. Recall that a fact has one positive literal (i.e., atomic formula) and no negative literals. A fact generally has the form:

\[ p(t_1, t_2, \ldots, t_n). \]

where \( p \) is a predicate symbol and \( t_1, t_2, \ldots, t_n \) are called terms. A fact is simply an assertion that it must be taken to hold true. Please note the period at the end of the fact.

For example, the fact “\( p1(a, b) \)” asserts that the predicate \( p1 \) holds between objects denoted by \( a \) and \( b \).

The terms (to be discussed later in class) may be arbitrarily complex.

\[ p(t_1, t_2, \ldots, t_n). \]

\[ \text{point}(x,y,z). \]
\[ s(np(john), vp(v(likes), np(mary))). \]
are examples of facts.

The following is a hypothetical database of facts:

```
likes(jim, car). /* here jim and car refer to two objects in the world */
likes(jim, apple).
likes(jim, books).
likes(mary, apple).
owns(jim, chevy).
owns(jim, apple).
owns(jim, book).
```

**Query** – Recall a query is a Horn clause with no positive literal. Actually, queries are questions you ask at the “prompt” by listing a sequence of atomic formulae separated by comma. Typically we type the facts and rules (i.e., the program) into a file, and the file is “consulted” which is analogous to being loaded in lisp.

Note that despite the fact that we say facts, rules and queries are Horn clauses, we never use the symbol for negation. Commas separating the goals in the body of a rule or in a query can be read as conjunctions; :- can be read as “if”.

```
?- owns(mary, car).
```

This query is asking if the “owns” relationship exists between the objects mary and car. The question being queried is whether “does Mary own a car” follows from (is a consequence of) the program, a set of facts and rules.

In order to determine the answer to this question, prolog searches through the program database in a top-down (left to right) fashion seeing if there are any fact (or rule) that matches the query goal.

Notice that “no” means “can not be established” as prolog makes the so-called “closed world assumption”.

Now things get more interesting when we introduce variables into our questions. Variables must be marked in some way – typically constants and predicate symbols start with a small letter, variables with capital letters.

```
?- likes(jim, X).
X= car ; /* note that the ; (which the user types in) asks that
    the interpreter continue and attempt to prove it again */
X= apple;
X= books;
no
```

There is a way of asking questions when one doesn’t care about the value of the variable is. This is done by using an “anonymous variable” – its value is not reported and its various occurrences need not be instantiated the same way each time.
So, to ask if “jim likes anything” type \texttt{likes(jim, \_)} at the prompt \texttt{?-}.

As noted before, a query could have more than one literal. When there are more than one literal, then we have \textit{conjunctive goals}.

\texttt{?- likes(jim, X), owns(jim, X)}

Note that this has the variable \texttt{X} in both conjuncts. Prolog will start at the top of the database, and attempt to find a value for \texttt{X} which makes the first goal in the query true. Once this is found, and \texttt{X} is instantiated, and prolog will start at the top of the database and attempt to prove the second conjunct true (with the given instantiation). If it cannot prove the second conjunct with the given instantiation of the variable, a failure will occur, causing the original goal to backtrack and be proved another way. Proving it another way will give a new value for \texttt{X}, and given that new value, the second conjunct will attempt to be proven by starting at the top of the database again. For example, given the above set of facts, in trying to establish the conjunctive goals, \texttt{X} is first instantiated as \texttt{car}. But the second goal can not be established with this instantiation. Prolog will backtrack leading to the second instantiation of \texttt{X} with \texttt{apple}.

Note that if the query had been

\texttt{?- owns(jim, X), likes(jim, X)}

then the answer \texttt{apple} would still be obtained. However, note that this time the interpreter does not even consider \texttt{car} as a possible answer (although it will consider \texttt{chevy} this time).

\textbf{Rules} – general statements about objects and relationships. Recall that a rule has one positive literal and one or more negative literal. Thus, a rule is said to have two parts – a head (which is the single positive literal) and a body (one or more negative literals separated by comma). \texttt{:-} is used to separate these two parts. Prolog interpreter interprets the rule as a statement that the head will hold if the body holds.

Prolog will do backward chaining in order to satisfy the head by attempting to satisfy each goal in the body. Some examples:

\texttt{p :- q, r, s.} \quad \text{Reads: p holds if q, r, and s are assumed to be true.}

\texttt{likes(jim, X) :- likes(X, books), likes(X, apple).}

The prolog interpreter attempts to satisfy the conjunction of goals in a left to right order to establish that head is satisfied. Therefore, if there is an \texttt{X} such that \texttt{likes(X, books)} can be established and for the same instantiation of \texttt{X}, \texttt{likes(X, apple)} can also be established, then the prolog interpreter assumes that the head \texttt{likes(jim, X)} has been established with that instantiation of \texttt{X}. Note the instantiation of \texttt{X} as \texttt{jim} satisfies the first goal. This instantiation also allows a successful satisfaction of the second goal. Hence, \texttt{likes(jim, jim)} can be satisfied from the above set of facts; on the other hand, \texttt{likes(jim, mary)} can not be established.
If the input goal matches the head of a rule, the instantiated variables which were required to match the head are made throughout the rule, then prolog goes about trying to satisfy the body by satisfying each of the goals in the body one at a time (and making the proper variable instantiations as it goes). If all of the goals in the body can be satisfied, the the head is said to be satisfied. If the query (input goal) was \texttt{likes(jim, mary)}, it matches the head of the rule with X being instantiated as \texttt{mary}. Thus prolog will attempt to satisfy the two goals in the body with this instantiation; i.e., \texttt{likes(mary, books)} and \texttt{likes(mary, apple)}. Note as the first one does not follow from the facts given, prolog interpreter will not even attempt to establish the second one, although it does follow from the given facts.

Consider another example. Suppose the following facts:

\begin{verbatim}
male(jeff).
male(fred).
female(mary).
parents(jeff, vince, jane).
parents(mary, vince, jane).
\end{verbatim}

We want to come up with a sister-of rule – sister-of(X,Y) would hold if X is a sister of Y.

- if X is female
- if X and Y have the same parents.

We could capture this in a prolog rule:

\begin{verbatim}
sister_of(X,Y) :- female(X),
    parents(X, F, M),
    parents(Y, F, M).
\end{verbatim}

Note here we are using a variable that is not in the head of the rule. So, when the goal is initially matched against the head, and the conjunction of goals is posted, F and M are not initially instantiated and will match against whatever is in the database for the instantiated X. Once the female(X) is satisfied (assuming that X wasn’t bound by matching the head), the first parent goal will be posted with X bound by with F and M unbound. Satisfying them might cause F and M to become bound, and the second parents clause will be posted with F and M bound (but Y need not be). If a suitable instantiation for Y can not be found, this will cause back-up through to the female clause to find other instantiations of the variables.

\begin{verbatim}
| ?- sister_of(X,Y).
X = mary
\end{verbatim}
Y = jeff;
X = mary
Y = mary;
no

| ?- trace(sister_of(X,Y)).
Call: sister_of(_281,_343) <enter=call, other=trace>: t
   !!! dynamic(sister_of(_281,_343))
Call: female(_281) <enter=call, other=trace>: t
   !!! dynamic(female(_281))
Exit: female(mary)
Call: parents(mary,_782,_783) <enter=call, other=trace>: t
   !!! dynamic(parents(mary,_782,_783))
Exit: parents(mary,vince,jane)
Call: parents(_343,vince,jane) <enter=call, other=trace>: t
   !!! dynamic(parents(_343,vince,jane))
Exit: parents(jeff,vince,jane)
Exit: sister_of(mary,jeff)
X=mary,
Y=jeff;

Redo: sister_of(mary,jeff)
   Redo: parents(jeff,vince,jane)
   Exit: parents(mary,vince,jane)
Exit: sister_of(mary,mary)
X=mary,
Y=mary;

Redo: sister_of(mary,mary)
   Redo: parents(mary,vince,jane)
   Fail: parents(_343,vince,jane)
   Redo: parents(mary,vince,jane)
   Fail: parents(mary,_782,_783)
   Redo: female(mary)
   Fail: female(_281)
   Fail: sister_of(_281,_343)
no

| ?- sister_of(Y, fred).
no
?- trace(sister_of(Y, fred)).
Call: sister_of(_281,fred) <enter=call, other=trace>: t
   !!! dynamic(sister_of(_281,fred))
Call: female(_281) <enter=call, other=trace>: t
!!! dynamic(female(_281))

Exit: female(mary)

Call: parents(mary,_787,_788) <enter=call, other=trace>: t
  !!! dynamic(parents(mary,_787,_788))

Exit: parents(mary,vince,jane)

Call: parents(fred,vince,jane) <enter=call, other=trace>: t
  !!! dynamic(parents(fred,vince,jane))

Fail: parents(fred,vince,jane)

Redo: parents(mary,vince,jane)

Fail: parents(mary,_787,_788)

Redo: female(mary)

Fail: female(_281)

Fail: sister_of(_281,fred)

no

?- trace(sister_of(mary,X)).

Call: sister_of(mary,_350) <enter=call, other=trace>: t
  !!! dynamic(sister_of(mary,_350))

Call: female(mary) <enter=call, other=trace>: t
  !!! dynamic(female(mary))

Exit: female(mary)

Call: parents(mary,_787,_788) <enter=call, other=trace>: t
  !!! dynamic(parents(mary,_787,_788))

Exit: parents(mary,vince,jane)

Call: parents(_350,vince,jane) <enter=call, other=trace>: t
  !!! dynamic(parents(_350,vince,jane))

Exit: parents(jeff,vince,jane)

Exit: sister_of(mary,jeff)

X=jeff;

Redo: sister_of(mary,jeff)

Redo: parents(jeff,vince,jane)

Exit: parents(mary,vince,jane)

Exit: sister_of(mary,mary)

X=mary;

Redo: sister_of(mary,mary)

Redo: parents(mary,vince,jane)

Fail: parents(_350,vince,jane)

Redo: parents(mary,vince,jane)

Fail: parents(mary,_787,_788)

Redo: female(mary)

Fail: female(mary)

Fail: sister_of(mary,_350)

no
?- trace(sister_of(mary,X)).
Call: sister_of(mary,_350) <enter=call, other=trace>: t
  !!! dynamic(sister_of(mary,_350))
Call: female(mary) <enter=call, other=trace>: t
  !!! dynamic(female(mary))
Exit: female(mary)
Call: parents(mary,_787,_788) <enter=call, other=trace>: t
  !!! dynamic(parents(mary,_787,_788))
Exit: parents(mary,vince,jane)
Call: parents(_350,vince,jane) <enter=call, other=trace>: t
  !!! dynamic(parents(_350,vince,jane))
Exit: parents(jeff,vince,jane)
Exit: sister_of(mary,jeff)
X=jeff

yes
?-
1 Processing Summary

1. Attempt to satisfy a goal $g_i$. In general this goal will be a member of a conjunction of goals we are attempting to satisfy in left to right order: $g_1, g_2, ..., g_{i-1}, g_i, g_{i+1}, ..., g_n$. Here, $g_1, g_2, ..., g_{i-1}$ have already been satisfied, and we are currently attempting to satisfy $g_i$.

   We start at the top of our database and:

   (a) The goal may be matched by matching (or unifying) with a fact or a rule head in the database. Mark that place in the database and instantiate any variables occurring in the remainder of the goal (i.e., $g_i, g_{i+1}, ..., g_n$) and in the rule body that were bound during that unification process. If the match took place with a rule head, attempt to satisfy the rule. If this succeeds or if the original match was with a fact, then attempt to satisfy $g_{i+1}$.

   (b) If no match is found (or all such matches with rule heads failed) then goal $g_i$ has failed. Attempt to resatisfy goal $g_{i-1}$.

2. Attempt to re-satisfy a goal. Return to uninstantiated any variables which were bound when goal was previously satisfied. Resume search of the database from where the place marker was previously put.

   In general, you want to be careful that you don’t make circular definitions:

   \[
   \begin{align*}
   \text{parent}(X,Y) & :\!\!\!: \text{child}(Y,X). \\
   \text{child}(X,Y) & :\!\!\!: \text{parent}(Y,X).
   \end{align*}
   \]

   You want to make sure that you will get closer and closer to a goal. Be aware of left recursion!

   \[
   \begin{align*}
   \text{person}(X) & :\!\!\!: \text{person}(Y), \text{parent}(X,Y). \\
   \text{person}(\text{adam}).
   \end{align*}
   \]

   Notice that because of the ordering here, this would get tied up in the rule. As a result, prolog will not be able to establish the goal person(adam) despite the fact that it is stated in the program.